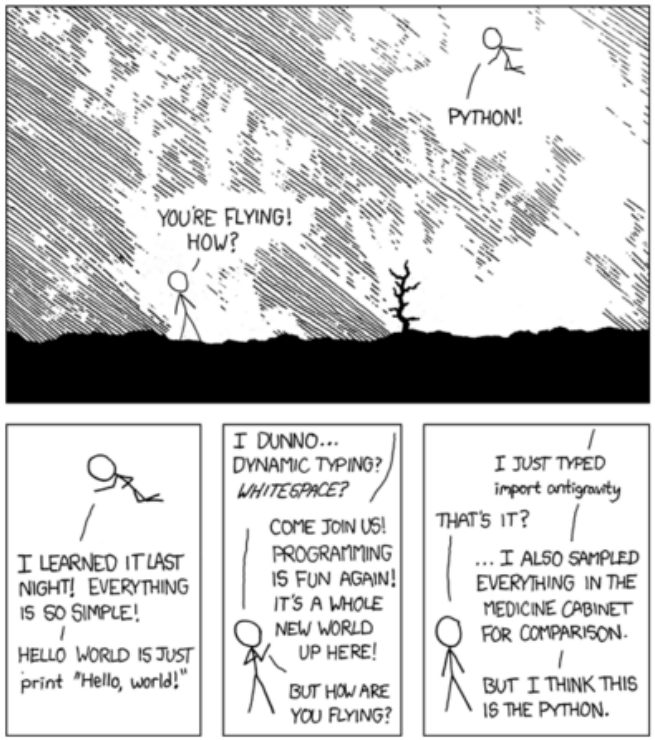
Qui veut du Python ?

Anthony VEREZ (netantho@minet.net)

Décembre 2011

Pourquoi Python ?



Quelques informations

- Langage interprété, interpréteur interactif avec la commande *python*
- 8ème langage le plus utilisé
- Utilisé par la NASA, Google, Battlefield 2, ...
- Deux branches : 2.x (actuellement 2.7.2) depuis 2001 et 3.x (actuellement 3.2.2) depuis 2009. Quand ce n'est pas précisé, je vais travailler ici avec **Python 2.x**
- Indentation des blocs avec espaces obligatoire
- Librairie standard riche
- Programmation orienté objet
- Python est déjà installé sur la plupart des distributions Linux ;)

Indentation en C

Lâchez le troll !

```
1  #include <stdio.h>
2
3  /* Donne FALSE */
4  int main()
5  {
6      if (1)
7          if (0)
8              printf("TRUE\n");
9      else
10         printf("FALSE\n");
11     return 0;
12 }
```

Indentation en Python

C'est mieux ! :P

```
1  foo = True
2  str = "foo"
3  if foo:
4      if str == "bar":
5          print "Bar"
6      else:
7          print "Foo"
8      print "Foo !!!"
9  else:
10     print "Bar !!!"
```

Typage Fort

Python est un langage à typage fort dynamique.

- Les types sont déterminés à l'exécution
- La compilation et l'éxecution détectent les erreurs de typage
- Pas de conversion implicite de types, sauf un peu pour les nombres (!= PHP)

Je veux de la doc !

- Documentation officielle
- Plongez au cœur de Python
- AFPY Association Francophone Python
- Présentation de Boris Feld, cette présentation en est inspirée
- Les bonnes pratiques
- PEP Python Enhancement Proposals

```
1 >>> help(fonction)
2 >>> object.__dict__
```

Types de variables

```
>>> type(42)
<type 'int'>
>>> type(42.) #42.==42.0
<type 'float'>
>>> type("texte")
<type 'str'>
```

```
>>> type(True)
<type 'bool'>
>>> type(0L)
<type 'long'>
>>> type(1+1j)
<type 'complex'>
```

Pas d'équivalent de *char*, on n'a que des chaînes de caractères

```
>>> type('a')
<type 'str'>
```

Conversion de types (typecasting) avec *int(foo)*, *str(foo)*, ...

Chaînes de caractères

- Concaténation avec + ou *join(liste)*

```
>>> "foo "+str(42)+" bar"
'foo 42 bar'
>>> couleurs = ['rouge', 'bleu', 'vert', 'jaune']
>>> ' '.join(couleurs) # pour liste
'rouge bleu vert jaune'
>>> x = 'string sur plusieurs \
>>> ... lignes'
>>> print x
'string sur plusieurs lignes'
>>> print "Vive\nINTech"
Vive
INTech
```

"Printf"

```
>>> print "%.5d euros" % (3*10)
00030 euros
>>> print "%d + %d = %d" % (2, 3, (2+3))
2 + 3 = 5
```

Opérations arithmétiques

- Opérateurs : + - * / %
- Une opération sur deux nombres dans deux ensembles différents donne le résultat dans le plus "grand" (au sens de l'inclusion) ensemble

```
1 >>> 2*5.
2 10.0
3 >>> 2*5j
4 10j
```

Affectations

```
1 >>> x = 0
2 >>> x
3 0
4 >>> y, z = 1, 0
5 >>> y
6 1
7 >>> z
8 0
```

Booléens

Valeurs considérées comme fausses :

- False
- None
- 0, 0.0, 0L, 0j
- ""
- (), [ ], { } on en reparle ;)

```
1 >>> x or y # OU logique (inclusif)
2 >>> x and y # ET logique
3 >>> not x # NON logique
```

Listes

Structure ordonnée de taille dynamique contenant une collection d'objets de types qui peuvent être différents

```
>>> x = []
>>> type(x)
<type 'list'>
>>> x = [1, 2, 'a', 'c']
>>> x
[1, 2, 'a', 'c']
>>> x[3]
'c'
>>> list('Plop')
['P', 'l', 'o', 'p']
```

Listes - Fonctions utiles

```
>>> x.append(5)
>>> x
[1, 2, 'a', 'c', 5]
>>> x.remove('a')
>>> x
[1, 2, 'c', 5]
>>> len(x) # taille
4
>>> l = [1, 2, 1, 1]
>>> l.count(1)
3
>>> l.count(2)
1
>>> l[1:]
[2, 1, 1]
>>> l[::-1]
[1, 1, 2, 1]
>>> # Premier indice
>>> # d'un element
>>> l.index(1)
0
```

Tuples

Comparables aux listes à la différence que les valeurs des tuples ne sont pas modifiables

```
>>> x = (1,2)
>>> x
(1, 2)
>>> type(x)
<type 'tuple'>
>>> x[1]
2
>>> x[:1]
(1,)
>>> x[1::2]
(2,)
>>> # pas tuple
>>> x = (1)
>>> x
1
>>> # tuple
>>> x = (1,)
>>> x
(1,)
>>> x[0]
1
>>> x[0] = 2
Traceback (most
recent call last):
File "<stdin>",
line 1, in <module>
TypeError: 'tuple'
object does not
support item
assignment
```

Tuples - Fonctions utiles

```
>>> l = (1, 2, 2, 3, 3, 3)
>>> l.count(2)
2
>>> l.count(3)
3
>>> l = (1, 2, 1)
>>> l.index(1)
0
>>> l.index(2)
1
>>> len(l)
3
```

Indexation négative

```
1 >>> x = [1, 2, 3, 4, 5]
2 >>> x[-1]
3 5
4 >>> x[-2]
5 4
6 >>> x[-5]
7 1
```

Dictionnaires

Accès aux éléments non pas par des index mais par des clés (chaîne de caractères, tuple, ...), genre de "tableau associatif".

- Clés non modifiables
- Structure non ordonnée
- Une clé pointe vers une seule valeur (qui peut néanmoins être une liste, un tuple ou un dictionnaire)

Dictionnaires

```
>>> positions = {}
>>> positions
{}
>>> positions = {(0,0): True, (0,1): False}
>>> positions
{(0, 1): False, (0, 0): True}
>>> positions[(0,0)]
True
>>> positions[(1,1)] = 'a'
>>> positions
{(0, 1): False, (0, 0): True, (1, 1): 'a'}
>>> positions[(0,1)] = True
>>> positions
{(0, 1): True, (0, 0): True, (1, 1): 'a'}
```

Dictionnaires - Fonctions utiles

```
>>> len(positions)
3
>>> positions.values()
[True, True, False]
>>> positions.keys()
[(0, 1), (0, 0), (1, 1)]
>>> del positions[(0,1)]
>>> positions
{(0, 0): True, (1, 1): 'a'}
>>> (1,1) in positions
True
```

Les itérables

Sont itérables

- Les chaînes de caractères
- Les tuples, listes et dictionnaires
- Les fichiers (sur les lignes)

Attention, sur les dictionnaires, les éléments itérés sont les clés alors que pour listes et tuples ce sont les valeurs

Les itérables

```
>>> ch = 'coucou'
>>> 'o' in ch
True
>>> map(lambda x: x+'a', ch)
['ca', 'oa', 'ua', 'ca', 'oa', 'ua']
>>> dict = {'c': 'b', 'b': 'c', 'a': 'd'}
>>> sorted(dict)
['a', 'b', 'c']
>>> sorted(dict.values())
['b', 'c', 'd']
>>> reversed(ch)
<reversed object at 0x1800a90>
>>> ''.join(list(reversed(ch))) # ne caste pas en str
'uocuoc'
```

Conditions

Les opérateurs de comparaison

- ==
- !=
- <=, >=, <, >

```
1 >>> x = 0
2 >>> if x > 0:
3 ...     print "Plus grand que 0"
4 ... elif x < 0:
5 ...     print "Plus petit que 0"
6 ... else:
7 ...     print "Egal 0"
8 ...
9 Egal 0
```

Boucle for

- Sur itérable

```
>>> x = (1, 2, 3)
>>> for i in x:
...     print i
...
1
2
3
```

- utilisation "Classique"

```
>>> range(5)
[0, 1, 2, 3, 4]
>>> range(2, 7)
[2, 3, 4, 5, 6]
>>> for i in range(5):
...     print i,
...
0 1 2 3 4
>>> for i in range(5,2,-1):
...     print i,
...
5 4 3
```

Boucle while

```
>>> b, i = True, 0
>>> while b:
...         print i,
...         i += 1
...         if i >= 5:
...             b = not b
...
0 1 2 3 4
```

break et continue

```
>>> compteur = 0
>>> while True:
...         print compteur,
...         compteur += 1
...         if compteur >= 5:
...             break # Arrete l'execution de la boucle
...
0 1 2 3 4
>>> for x in range(10):
...         if x % 2 == 0:
...             continue # Passe au bloc suivant
...         print x
...
1 3 5 7 9
```

Entrées utilisateur

```
>>> s = raw_input('Vous suivez encore ? ')
Vous suivez encore ? oui
>>> s
'oui'
```

```
$ echo "import sys" >> cli.py
$ echo "print sys.argv" >> cli.py
$ python cli.py INTech Powa
['cli.py', 'INTech', 'Powa']
```

Fonctions

```
>>> def hello_world():
...     print "Hello World!"
...
>>> hello_world()
Hello World!
>>> def fonction(arg1, arg2, arg3 = "val3", arg4 = "val4
...     return (arg1, arg2, arg3, arg4)
...
>>> fonction("val1", "val2")
('val1', 'val2', 'val3', 'val4')
>>> fonction(arg2 = "val1", arg1 = "val2")
('val2', 'val1', 'val3', 'val4')
>>> fonction("val1", "val2", arg4 = "val1")
('val1', 'val2', 'val3', 'val1')
```

Démo !!!

Challenge

Afficher les 10 derniers tweets contenant 'python' avec l'utilisateur associé et la date

Corrigé

```
1   import urllib
2
3   url = urllib.urlopen("http://search.twitter.com/search.json?q=python")
4
5   import json
6
7   monjson = json.loads(url.read())
8   for elem in monjson['results']:
9       print elem['from_user_name']+" le "+elem['created_at']
10      print elem['text']+"\n"
```

Tadam ! Rapide, non ?

Les fichiers

```
>>> f = open('fichier', 'w')
>>> f.write("Coucou")
>>> f.close()

>>> # Permet de fermer le fichier automatiquement
>>> # apres traitements
>>> with open('fichier', 'r') as f:
...     print f.readlines()
['Quoi']
>>> with open('fichier', 'r') as f:
...     print f.read()
'Quoi'
```

Pour les chemins, voir le module os

Idées principales et jargon

Un *objet* en programmation est une structure de données qui contient à la fois :

- Des *attributs* : données sous forme de variables
  - de classe : propres à une classes et communs à tous les objets de même type
  - d'instance : propres à un objet
- Des *méthodes* : des moyens de traiter ces données sous forme de fonctions

Un objet est une instance d'une *classe*
Le *constructeur* est la méthode qui est appelée à la création d'un objet

Classes

```
>>> class Robot:
...     def __init__(self, couleur, roues): # constructeur
...         self.couleur = couleur # attribut d'instance
...         Robot.roues = roues # attribut de classe
...     def changerCouleur(self, couleur):
...         self.couleur = couleur
...
>>> bot1=Robot("rouge",2)
>>> bot1.roues
2
>>> bot1.couleur
'rouge'
>>> bot1.changerCouleur("jaune")
>>> bot1.couleur
'jaune'
>>> bot2=Robot("bleu",3)
>>> bot1.roues
3
>>> bot2.couleur
'bleu'
```

Héritage

```
>>> class A:
...     a = 'A'
>>> class B:
...     b = 'B'
...
>>> # la classe C herite des classes A et B
>>> class C(A,B):
...     pass
>>> c = C
>>> c.a
'A'
```

On peut aussi hériter de méthodes de la même façon

Surcharge d'opérateurs

Définir les opérateurs entre des objets de même types. Il faut définir des méthodes spéciales dans les classes qui doivent retourner des booléens :

- object.__lt__(self, b) : $a < b$
- object.__le__(self, b) : $a <= b$
- object.__eq__(self, b) : $a == b$
- object.__ne__(self, b) : a ! = b
- object.__gt__(self, b) : $a > b$
- object.__ge__(self, b) : $a >= b$

On a aussi des méthodes spéciales pour les opérations arithmétiques.

Modularité

Un programme python peut s'organiser sous forme de modules.
Créons le module script :

script.py

```
1  var = 3
2  def ma_fonction():
3      print "ma fonction"
4  class C:
5      pass
6  print "mon module"
```

Dans le même dossier, lançons l'interpréteur intéractif :

```
1  >>> import script # module execute a l'importation
2  mon module
3  >>> locals()
4  {'__builtins__': <module '__builtin__' (built-in)>, '__name__':
5  '__main__', 'script': <module 'script' from 'script.py'
```

Modularité

Pour éviter ce genre de chose, on peut faire ça :

modularite_execution.py

```
1 print("Nom : " + __name__)
2 if __name__ == "__main__":
3         print "Execution directe"
4 if __name__ == "modularite_execution":
5         print "Importation"
```

```
1 $ python modularite_execution.py
2 Nom : __main__
3 Execution directe
```

```
1 >>> import modularite_execution
2 Importation
```

Modularité - Packages

On peut avoir un dossier complet pour module avec chaque fichier en sous-module.
Fichier obligatoire : pack/__init__.py

```
1 print "Importation du package pack"
```

pack/test.py

```
1 def f():
2     print "beuh !"
```

```
1 >>> import pack
2 Importation du package pack
3 >>> import pack.test
4 >>> pack.test.f()
5 beuh !
```

Modularité - Sélection fine

Je ne veux qu'une variable, fonction ou classe

```
>>> from pack.test import f
>>> pack
NameError: name 'pack' is not defined
>>> f
<function f at 0x7f09052ab5f0>
```

Tests unitaires

Un test unitaire permet de tester un programme en vérifiant généralement les retours de fonctions par rapport à des arguments données en guise de test
Des tests ? Pourquoi faire ?

- Maîtriser sa peur de codeur en vérifiant que le programme fonctionne
- Connaître rapidement l'état de dégradation de son code :P
- Ne pas faire du refactoring à l'aveuglette
- Améliorer la qualité de son programme
- Avoir un robot qui roule le jour de la coupe :D

Nose

Utilisation du module nose

Doc officielle (en)

Didacticiel (en)

```
1 $ sudo apt-get install python-nose
```

Pourquoi Nose ?

- Le module *unittest* existe dans la librairie standard mais il lui manque quelques fonctionnalités intéressantes
- Des tests qui s'écrivent facilement et rapidement dans une syntaxe python
- Couverture de code
- Configuration fine, plugins
- Performance : plugin pour faire du multithreadé

Utilisation basique de Nose

demo_tests.py

```
1  import re
2  EMAIL_REGEX = r'[\S.]+@[\S.]+'
3
4  class testEmail:
5      def test_email_regex(self):
6          assert re.match(EMAIL_REGEX, 'test@mail.ru')
7          assert not re.match(EMAIL_REGEX, 'test@where')
```

Utilisation basique de Nose

```
$ nosetests demo_tests.py
F
======================================================================
FAIL: demo_tests.testEmail.test_email_regex
----------------------------------------------------------------------
[...]
    assert not re.match(EMAIL_REGEX, 'test@where')
AssertionError

----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (failures=1)
```

Téléchargeur sans threads

```
1   import urllib
2   import time
3
4   def get_file(url):
5       mytime = time.ctime()
6       f = urllib.urlopen(url)
7       contents = f.read()
8       f.close()
9       print mytime+" "+url+" OK"
```

```
10
11  time_ini = time.ctime()
12
13  monurl = "http://dl.google.com/android/android-
14  for url in [monurl]*5:
15      get_file(url)
16
17  print time_ini
18  print time.ctime()
```

```
1   Sun Dec 11 23:58:00 2011 http://dl.google.com/android/android-sdk_r3-linux.tgz OK
2   Sun Dec 11 23:58:05 2011 http://dl.google.com/android/android-sdk_r3-linux.tgz OK
3   Sun Dec 11 23:58:06 2011 http://dl.google.com/android/android-sdk_r3-linux.tgz OK
4   Sun Dec 11 23:58:08 2011 http://dl.google.com/android/android-sdk_r3-linux.tgz OK
5   Sun Dec 11 23:58:09 2011 http://dl.google.com/android/android-sdk_r3-linux.tgz OK
6   Sun Dec 11 23:58:00 2011
7   Sun Dec 11 23:58:11 2011
```

Téléchargeur avec threads

```
1  import urllib, threading
2  import time
3
4  class FileGetter(threading.Thread):
5      def __init__(self, url):
6          self.url = url
7          threading.Thread.__init__(self)
8
9      def get_result(self):
10         return self.result
11
12     def run(self):
13         mytime = time.ctime()
14         f = urllib.urlopen(url)
15         contents = f.read()
16         f.close()
17         print mytime+" "+url+" OK"
18
19 time_ini = time.ctime()
20 tab = []
21 monurl = "http://dl.google.com/android/android-
22 for url in [monurl]*5:
23     tab.append(FileGetter(url))
24     tab[len(tab)-1].start()
25 for thread in tab:
26     thread.join()
27 print time_ini
28 print time.ctime()
```

```
1  Sun Dec 11 23:57:56 2011 http://dl.google.com/android/android-sdk_r3-linux.tgz OK
2  Sun Dec 11 23:57:56 2011 http://dl.google.com/android/android-sdk_r3-linux.tgz OK
3  Sun Dec 11 23:57:56 2011 http://dl.google.com/android/android-sdk_r3-linux.tgz OK
4  Sun Dec 11 23:57:56 2011 http://dl.google.com/android/android-sdk_r3-linux.tgz OK
5  Sun Dec 11 23:57:56 2011 http://dl.google.com/android/android-sdk_r3-linux.tgz OK
6  Sun Dec 11 23:57:56 2011
7  Sun Dec 11 23:58:04 2011
```

Améliorations à apporter

De nombreux d'autres problèmes sont à gérer :

- Gérer l'accès concurrent à une même ressource
- Mettre une file d'attente (voir de priorité) pour limiter à sa guise le nombre de threads
- Utiliser le module multiprocessing pour faire du multi-cœurs
- ...

socat

OMG J'émule une liaison série

Le site officiel (en)

```
1 $ sudo apt-get install socat
2 $ sudo apt-get install python-serial
3 $ socat PTY,link=./ptyp1,b9600 PTY,link=./ptyp2,b9600
```

ptyp1 et ptyp2 sont des pseudo terminaux reliés virtuellement par une liaison série.

Pyserial

Ouvrir deux autres terminaux (dans le même dossier) et lancer *$ sudo python*

```
>>> import serial
>>> ser = serial.Serial('./ptyp1')
>>> while 1:
...     ser.readline()
...
'INTech va tout faire peter !\n'
```

```

>>>import serial
>>>ser = serial.Serial('./ptyp2')
>>>ser.write("INTech va tout faire peter !\n")
29
```

```
>>> # debit de Baud de 19200, timeout 1s
>>> ser = serial.Serial('./ptyp1', 19200, timeout=1)
>>> ser.read() # Lire un bit
>>> ser.read(10) # Lire 10 bits
>>> ser.close # fermer la liaison proprement
```

Présentation de PyPy

PyPy est une implémentation de python (2.7.1 pour l'instant) destinée à accélérer, en moyenne 3x plus vite, l'éxecution d'un programme en python (**Attention** : il faut tester au cas par cas)
C'est un programme python (!!) qui fait de la compilation à la volée (JIT Compilation : Just-In-Time Compilation)
Télécharger PyPy
Choisir "JIT Compiler" version
Dézipper le tout, le programme est dans pypy-1.7/bin/pypy
Créer un lien symbolique de pypy (avec le chemin absolu) vers /usr/bin

Pour moi qui code, quelles sont les différences ?

- lancer pypy et non python
- Sauf pour des extensions exotiques, persos, ... les principales fonctionnalités sont gérées à l'identique
- Sur ma distribution Linux, mon Mac OS X tout va bien. Sur Windows, c'est un peu moins stable.
- Ce qui **ne marche pas** : trac, Zope, Genshi, matplotlib :(, NumPy (en cours) :(, gmpy, SciPy :(, SAGE, Bazaar, Subversion, SCons, pygame :(, pyGTK, Tkinter, Cython, Pyrex, SWIG, py2exe, Fabric, Pyro
- Ce qui **marche** : Django, Pylons, Pyramid, Twisted, IPython, Scapy :), SQLAlchemy, pymongo, PyMySQL, pysqlite, redis, PIL, mechanize :), mock, nose :), Selenium Python Client Driver, unittest2

Print

doc

```
>>> print "The answer is", 42 # Python 2.x
The answer is 42
>>> print("The answer is", 42) # Python 3.x
The answer is 42
>>> print(1,2) # Python 2.x
(1,2)
>>> print(1,2) # Python 3.x
1 2
```

Les Dictionnaires

```
>>># Python 2.x
>>>dict = {(1,2): "truc"}
>>>dict.keys()
[(1, 2)]

>>> # Python 3.x
>>> dict = {(1,2): "truc"}
>>> dict.keys()
dict_keys([(1, 2)])
>>> list(dict.keys())
[(1, 2)]
```

Idem pour *dict.items()*, *dict.values()*

map()

```
>>> # Python 2.x
>>> map(lambda x:x**2, [1,2])
[1, 4]

>>> # Python 3.x
>>> map(lambda x: x**2, [1,2])
<map object at 0x2327590>
>>> list(map(lambda x: x**2, [1,2]))
[1, 4]
```

Beaucoup d'autres changements moins importants.
Globalement, Python 3.x est plus orienté objet.
Des changements introduits dans Python 3.x sont peu à peu introduits dans Python 2.x avec rétrocompatibilité.

Listes compréhensives

```
>>> [n ** 2 for n in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> [n ** 2 for n in range(10) if n % 2]
[1, 9, 25, 49, 81]
```

Ce qui n'a pas été abordé

Voir la superbe présentation de Boris Feld

- Gestion des exceptions
- Les ensembles
- *args et **kwargs
- Callbacks
- Fonctions anonymes/lambda (entrevu)
- Fonction filter
- Les générateurs (le robot en aura peut-être besoin, à voir)
- Les décorateurs

Ce qui n'a pas été abordé

Voir sur le Net ou venir me voir

- Notions intermédiaires et avancée de la programmation orientée objet
- Gestion du verouillage, des files d'attente/de priorité pour les threads
- Utilisation intermédiaire et avancée de Nose
- Programmation avec les sockets (mais y'avait un club code dessus :P)

Licence